Recursion And BackTracking

Any function which calls itself is called *recursive*. A recursive method solves a problem by calling a copy of itself to work on a smaller problem. This is called the recursion step. The recursion step can result in many more such recursive calls. It is important to ensure that the recursion terminates.

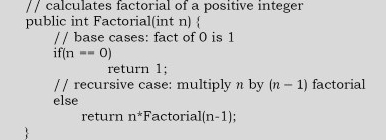
Recursive code is generally shorter and easier to write than iterative code. Generally, loops are turned into recursive functions when they are compiled or interpreted.

Example:

As an example consider the factorial function: *n*! is the product of all integers between n and 1. The definition of recursive factorial looks like:

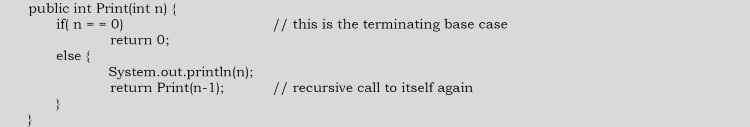


This definition can easily be converted to recursive implementation. Here the problem is determining the value of *n*!, and the subproblem is determining the value of (*n* – *l*)!. In the recursive case, when *n* is greater than 1, the function calls itself to determine the value of (n –*l*)! and multiplies that with *n*. In the base case, when *n* is 0 or 1, the function simply returns 1. This looks like the following:



**Recursion and Memory (Visualization)**

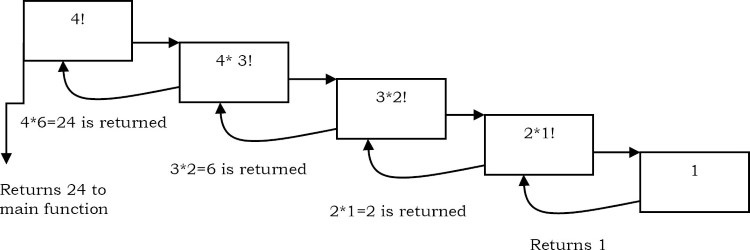
Each recursive call makes a new copy of that method (actually only the variables) in memory. Once a method ends (that is, returns some data), the copy of that returning method is removed from memory. The recursive solutions look simple but visualization and tracing takes time. For better understanding, let us consider the following example.



For this example, if we call the print function with n=4, visually our memory assignments may look like:



Now, let us consider our factorial function. The visualization of factorial function with n=4 will look like:



**Recursion versus Iteration**

**Recursion**

1. Terminates when a base case is reached.
2. Each recursive call requires extra space on the stack frame (memory).
3. If we get infinite recursion, the program may run out of memory and result in stack overflow.
4. Solutions to some problems are easier to formulate recursively.

**Iteration**

1. Terminates when a condition is proven to be false.
2. Each iteration does not require any extra space.
3. An infinite loop could loop forever since there is no extra memory being created.
4. Iterative solutions to a problem may not always be as obvious as a recursive solution.

Notes on Recursive:

* Recursive algorithms have two types of cases, recursive cases and base cases.
* Every recursive function case must terminate at a base case.
* Generally, iterative solutions are more efficient than recursive solutions [due to the overhead of function calls].
* A recursive algorithm can be implemented without recursive function calls using a stack, but it’s usually more trouble than its worth. That means any problem that can be solved recursively can also be solved iteratively.
* For some problems, there are no obvious iterative algorithms.
* Some problems are best suited for recursive solutions while others are not.

**Example Algorithms of Recursion**

* Fibonacci Series, Factorial Finding
* Merge Sort, Quick Sort
* Binary Search
* Tree Traversals and many Tree Problems: InOrder, PreOrder PostOrder
* Graph Traversals: DFS [Depth First Search] and BFS [Breadth First Search]
* Dynamic Programming Examples
* Divide and Conquer Algorithms
* Towers of Hanoi
* Backtracking Algorithms [we will discuss in next section]